STRUCTURAL REDUNDANCY FOR DETECTION AND CORRECTION OF ERRORS IN DATA STRUCTURES

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

By

R. MURALIDHAR

to the

COMPUTER SCIENCE PROGRAMME

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

APRIL, 1983

A82856

Fron. D.B

CERTIFICATE

This is to certify that this work on 'STRUCTURAL REDUNDANCY FOR DETECTION AND CORRECTION OF ERRORS IN DATA STRUCTURES' has been carried out by Mr. R. Muralidhar under my supervision, and it has not been submitted elsewhere for a degree.

S C Seth

Dr. S.C. Seth
Visiting Associate Professor
Computer Science Programme
Indian Institute of Technology
KANPUR.

IIT Kanpur.
29/4/83

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and gratitude to Dr. Sharad C. Seth for his able guidance and suggestions throughout the course of this work.

Thanks are also due to those who, directly or indirectly, helped in the preparation of this thesis.

IIT Kanpur R. MURALIDHAR
April , 1983

ABSTRACT

A study of errors occurring in stored data structures is made here. We restrict ourselves to structural errors. Structural errors are those changes which modify one or more nondata fields of nodes in a data structure instance. An axiomatic model for an implementation of a data structure is proposed for analysing its error detection capabilities. Some commonly used data structures are analysed using this model. A systematic method is suggested for deriving detection and diagnosis procedures for structural errors from the axiomatic descriptions. The problem of synthesis of data structures with a view to increase robustness is also dealt with.

CONTENTS

1. INTRODUCTION

1.1 Overview of the problem :

Software reliability presupposes the reliability of stored data structures upon which the programs operate. However, the stored data is prone to hardware and software errors. Redundancy is the key to robustness of data. Some portions of the data could be classified as the semantics or 'real data' while others are used to indicate the relationship between the various items in the data, classified as the structural information. In this study, we concentrate only on structural redundancy, which might be present for reasons other than reliability too. The semantics, is to be protected by using proper error detection/correction codes for storing the data.

1.2 What is a structural error?

The nondata or structural fields in a data structure are included to enable and simplify access to nodes and to speed up processing involved in performing standard operations (include, delete, read etc.) on the data structure. Some typical instances of structural fields are pointer, count, structure identifier, index limits etc.

A structural error in a data structure is any unintentional change in the structural fields of the data. This causes a valid data structure instance to change to either a valid or an invalid instance. Typical examples of structural errors are 'lost pointers' (pointers going astray) and a change of an identifier field associated with a node or a stored count.

1.3 Sources of errors :

The model we will use concerns itself only with the errors (i.e., the effects of failures) and not with the sources of error. However, it is only with reference to the actual failures causing errors that the validity of the assumption made in the model can be justified.

The errors could be due to one or more of the following reasons :

- Hardware errors (in the processor or in the storage and transmission media)
- Errors in user access to data
- Original creation of the data structure itself can be wrong, due to some bugs in the creating program
- Abrupt stoppage of update routines (As in the case of a system crash).

1.4 Measures of Robustness :

We will be concerned only with the robustness of structural information. Commonly used measures of robustness are detectability and correctability. Detectability is the maximum number of structural errors that can be detected. In effect, it signifies the maximum number of changes in structural data that will not lead to a valid instance of the data structure. Correctability denotes the maximum number of detectable structural errors that can be corrected to yield a unique valid instance. Other measures also exist and a summary of these can be found in [1].

1.5 Survey of the work already done :

The need for reliable data has long been realised. As early as 1968, Lockemann and Knutsen [5] proposed a method for recovery of disk contents after a system crash. It was basically an introduction to the idea of adding an unique identifier field to each record, which was later used extensively by Taylor et al. [1], [6-8]. Error recovery using log files, audit files, or using before- and after-images are standard ways of guarding against errors in data bases [3]. Only those parts of the data which were accessed during certain interval of time are duplicated in the form of log files, transaction files or before-images. This method, therefore, constitutes limited duplication of data and will not guard against all possible errors.

The idea of adding structural redundancy in terms of additional pointers is of obscure origin. The standard doubly linked list has come to stay, though their extensive use was made first in some of the implementations of the list processing languages. Even then, additional fields were added, not as a means of redundancy, but as aids to operations on the data structures. Likewise, the addition of a count of the number of nodes is also an advantage at very little extra cost and has been used for long.

An analysis of the error detection and correction capabilities of data structures for structural errors was first carried out by Taylor, Morgan and Black [6]. Some novel storage structures with improved roubustness are suggested in [6], and some of the experimental results about the robustness of data structures are also provided therein. The performance measures are presented in [1]. The theoretic aspects of the error detection/correction capabilities of data structures in general are dealt with in [7]. Some detection procedures are also given in [6]. Correction algorithms for some of the standard structures, as also the principles underlying their development, are given in [8].

1.6 Purpose of current study :

A rigorous mathematical analysis about the detectability of data structures is contemplated. So far, no attempt has been made in this direction. The reason was, apparently, the lack of a proper model of data structures, except a graph model by Earley [2]. Therefore, a mathematical model for data structures is also proposed. Apart from this, the need for a systematic method of developing detection procedures for any arbitrary data structure was felt and an attempt is made in Section 4 towards this. Diagnosis and correction algorithms can also be developed likewise for some of the cases. Finally, the problem of designing robustness in storage structures is considered in Section 5, after introducing the redundancy functional to characterize the redundancies in a data structure.

2. A MODEL FOR DATA STRUCTURES

2.1 The need for a model :

We propose to do a rigorous analysis of data structures in terms of its error detecting capabilities. A mathematical model will be appropriate for this purpose. Let us consider what are the features of a data structure which the model should capture.

The model need not tell anything about the user operations on the data structure. We are only concerned with the consistency of representation at any instance of time, independent of how it was arrived at. So, the user operations on the data structures are of no concern to us. But the model has to bring out those features which characterise a particular implementation of data structure.

For example, a binary tree could be implemented either as linked structure (with LSON-RSON fields) or as a linear array. Though the difference in implementations is supposed to be transparent to the user, we expect that our model should preserve this difference, as the robustness depends very much on the implementation.

Thus, in effect, when we say a data structure, we also mean a particular way of implementing it, and the model should be able to differentiate not only among various data structures, but also among a variety of implementations of one data structure.

2.2 The Abstract Mathematical Model :

A structure is denoted by S, its nodes by N_1, N_2 ... etc. Normally, there is a distinguished node N_H in S, called the header of the structure. In some cases, there may be more than one header, N_{H1}, N_{H2} ... and all the headers are linked together. Each of the headers is individually accessible without having the need to chase'pointers.

Fields are associated with each node. The fields could be 'data fields' or 'structure fields'. Here, we concentrate only on structural fields. The fields associated with a node N_1 are denoted by $f_1(N_1)$, $f_2(N_1)$ etc. It should be noted that $f_1(N_1)$ is only a shorthand notation for the f_1 field of N_1 and should not be looked upon as a function with argument N_1 as such.

Nil is a standard value which can be associated with any link type field and no field can be associated with it. For convenience, we will use the following notations :

f_1^k (where $k \geqslant o$) indicates chasing the 'f_1' field k times

f_1^{-k} (where $k \lessdot o$) indicates that node from which the current node would be reached after chasing 'f_1' field k times.

Every data structure is represented by a set of axioms or assertions. The assertions represent only the static nature of the data structure and no mention is made about the user operations on the data structure.

2.3 Representation of some common data structures :

We now give examples of representing some of the standard data structures in terms of the above model.

2.3.1 SINGLY LINKED LIST (SLL)

Consider a non-redundant implementation of SLL which has a pointer field f_1 (Fig. 2.1). It is uniquely characterised by the following assertions :

Assertion 1 : (a) $N_1 \varepsilon S$ and $f_1(N_1) = N_2 \Rightarrow N_2 \varepsilon S$

(b) $\forall N_1, N_2 \varepsilon S$' $f_1^k(N_1) = N_2$ for some finite $k \geq 0$

The above assertions say that any node pointed to by another node belonging to the structure is also in the structure. Assertion 1(b) states that the list is circular.

2.3.2 Robust SINGLY LINKED LIST (r-SSLL) :

Suppose the above representation of a simple singly linked list (SLL) is extended by introducing an Identifier field (I_d) with each node. In addition, let us assume that

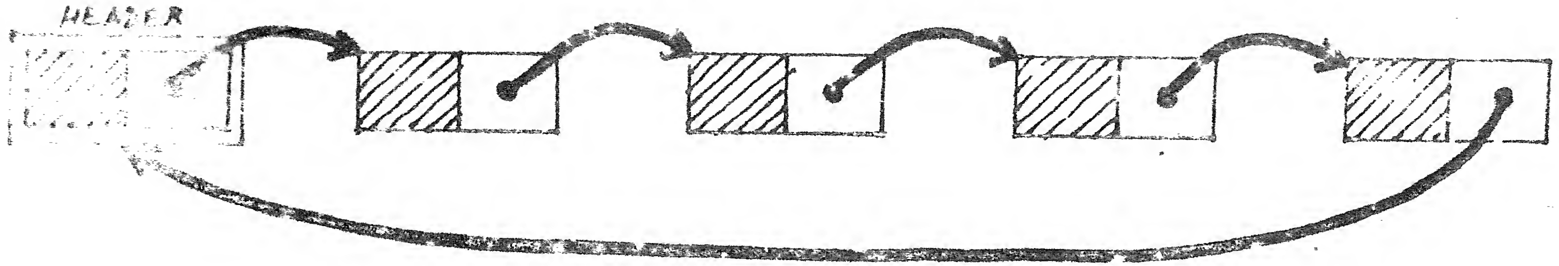


Fig. 2.1 A singly linked list

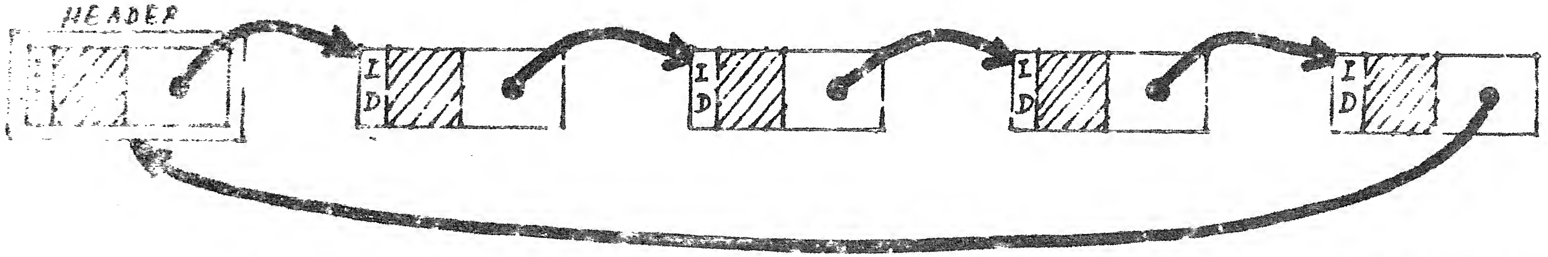


Fig. 2.2 A robust-singly linked list

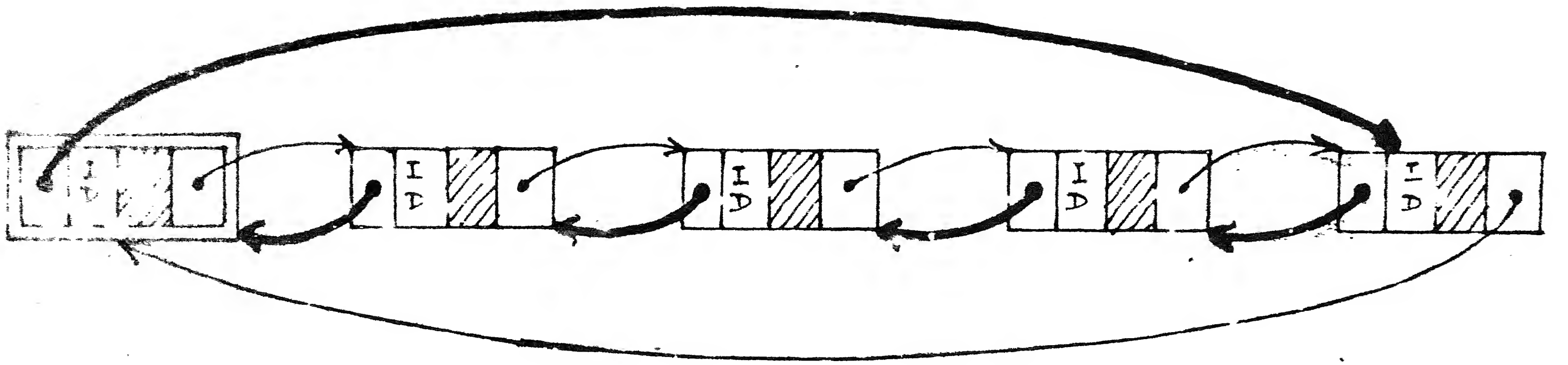


Fig. 2.3 A doubly linked list

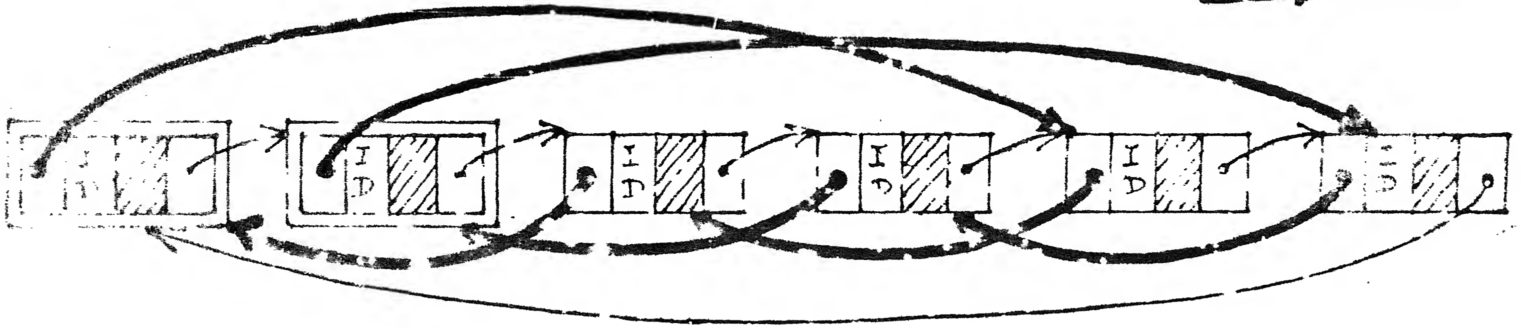


Fig. 2.4 A mod(2) doubly linked list

the header node has a 'count field' indicating the number of nodes in the structure (Fig. 2.2). We shall call this structure as robust SINGLY LINKED LIST (r-SLL). (It is possible that the header node might have a different node structure than the rest of the nodes).

This structure is characterised by the following assertions :

Assertion 1 : (a) $N_1 \varepsilon S$ and $f_1(N_1) = N_2 \Rightarrow N_2 \varepsilon S$

(b) $\forall N_1, N_2 \varepsilon S$, $f_1^k (N_1) = N_2$ for some finite $k \geqslant o$. This assertion says just those properties of r-SLL which are shared by SLL.

Assertion 2 : $f_1(N_1) = N_2 \Rightarrow I_d(N_1) = I_d(N_2)$

This is to say that if two nodes are 'connected' they have the same I_d value. Together with 1(a), this also implies that all the nodes in the structure have a unique Identifier value.

Assertion 3 : If m = count (S)

then $f_1^k(N_H) = N_H$ iff k = m+1

(for circular lists)

If m is the stored count of the number of nodes in the structure, then, starting from the header, we must reach back to the header, after following the f_1 field successively m+1 times. This should not happen for any other sequence of following the pointer field.

The significant fact here is that m is a known constant.

2.3.3 DOUBLY LINKED LIST (DLL) :

Now assume that, on the structure of a r-SLL, we make the following modification. Every node has an additional pointer field f_2 and it points to the predecessor of the node. (if we consider f_1 field to contain a value that defines the normal sequence of nodes). See Fig. 2.3.

This structure will be called the doubly linked list (DLL) and is characterised by the following assertions :

Assertion 1 : (a) $N_1 \varepsilon S$ and $f_1(N_1) = N_2 \Rightarrow N_2 \varepsilon S$

(b) $\forall N_1, N_2 \varepsilon S, f_1^k(N_1) = N_2$ for some finite $k \geqslant 0$

Assertion 2 : (a) $f_1(N_1) = N_2 \Rightarrow f_2(N_2) = N_1$

(b) $f_2(N_1) = N_2 \Rightarrow f_1(N_2) = N_1$

These just mention the fact that if node N_1 points to N_2 in the forward sequence (via f_1), then N_2 points to N_1 (via f_2), and vice versa.

Assertion 3 : (a) $f_1(N_1) = N_2 \Rightarrow I_d(N_1) = I_d(N_2)$

(b) $f_2(N_1) = N_2 \Rightarrow I_d(N_1) = I_d(N_2)$

Assertion 4 : If m = count (s)

then (a) $f_1^k(N_H) = N_H$ if: $k = m+1$

(b) $f_2^k(N_H) = N_H$ if: $k = m+1$

Assertions 3 and 4 are direct extensions of the corresponding assertions for r-SLL.

2.3.4 MODIFIED (2) DOUBLY LINKED LIST :

This is a novel doubly linked list suggested in [6]. Here, instead of the back pointer (content of f_2 field) pointing to the previous node, we make it point to the node preceding it. This will be referred to as the mod (2) DLL. The number 2 indicating the distance spanned by the back pointer (Fig. 2.4). Note that there are two headers N_{H1} and N_{H2}. In general, it will be required to have k headers for a mod (k) DLL.

This structure is uniquely characterised by the following assertions :

Assertion 1 : (a) $N_1 \in S$ and $f_1(N_1) = N_2 \Rightarrow N_2 \in S$

(b) $\forall N_1, N_2 \in S$, $f_1^k(N_1) = N_2$ for some finite $k \geqslant o$

This is same as for the DLL.

Assertion 2 : (a) $f_1^2(N_1) = N_2 \Rightarrow f_2(N_2) = N_1$

(b) $f_2(N_1) = N_3 \Rightarrow f_1^2(N_3) = N_1$

This brings out the essential difference between a DLL and the mod (2) DLL.

Assertion 3 : (a) $f_1(N_1) = N_2 \Rightarrow I_d(N_1) = I_d(N_2)$

(b) $f_2(N_1) = N_2 \not\Rightarrow I_d(N_1) = I_d(N_2)$

This is again same as for the DLL.

Assertion 4 : If m = count (s)

then

(a) $f_1^k(N_{H1}) = N_{H1}$ iff $k = m+2$

(b) $f_2^k(N_{H1}) = N_{H1}$

$\underline{\text{and}}$ $f_2^k(N_{H2}) = N_{H2}$

$$\text{iff} \begin{cases} k = \frac{m}{2} + 1 & \text{for } m \ \underline{\text{even}} \\ k = m+2 & \text{for } m \ \underline{\text{odd}} \end{cases}$$

Assertion 4(b) is a consequence of having two headers and it is necessary to consider the casesof even and odd values of the number of nodes separately.

2.3.5 SIMPLE BINARY TREE :

We consider the standard binary tree implementation with Left link and Right link fields (f_1 and f_2 respectively) to denote the left and right son.

It is characterised by the following assertions :

Assertion 1 : $\forall_{N1} \in S$, $N2 \neq nil$

$$\text{(a) } f_1(N_1) = N_2 \Rightarrow N_2 \in S$$

$$\text{(b) } f_2(N_1) = N_2 \Rightarrow N_2 \in S$$

This asserts that a node could be 'connected' to its parent either through the left or the right field.

Assertion 2 : (a) $f_1(N_1) = N_2 \Rightarrow f_1(N) \neq N_2$ <u>and</u> $f_2(N) \neq N_2$

for any $N \neq N_1$

(b) $f_2(N_1) = N_2 \Rightarrow f_1(N) \neq N_2$ <u>and</u> $f_2(N) \neq N_2$

for any $N \neq N_1$

$$\text{(c) } \left.\begin{array}{l} f_1(N) \neq N_H \\ f_2(N) \neq N_H \end{array}\right\} \text{ for any } N \in S$$

This asserts the acyclic property.

2.3.6 THREADED/CHAINED BINARY TREE :

The left and right link-fields of those nodes which contain logically null value do not contain much information. These fields, therefore, could be used to represent some other relationship among the nodes. The null right fields might be 'threaded' to point to its in-order successor and the null left fields can be 'chained' to the next such node in the in-order sequence whose left field also contains a logical nil value, as suggested in [7]. Tags t_1 and t_2 might be needed in this case to indicate whether the fields f_1 and f_2 contain normal pointers or not. Tag = 0 may indicate a normal pointer. Refer Fig. 2.5.

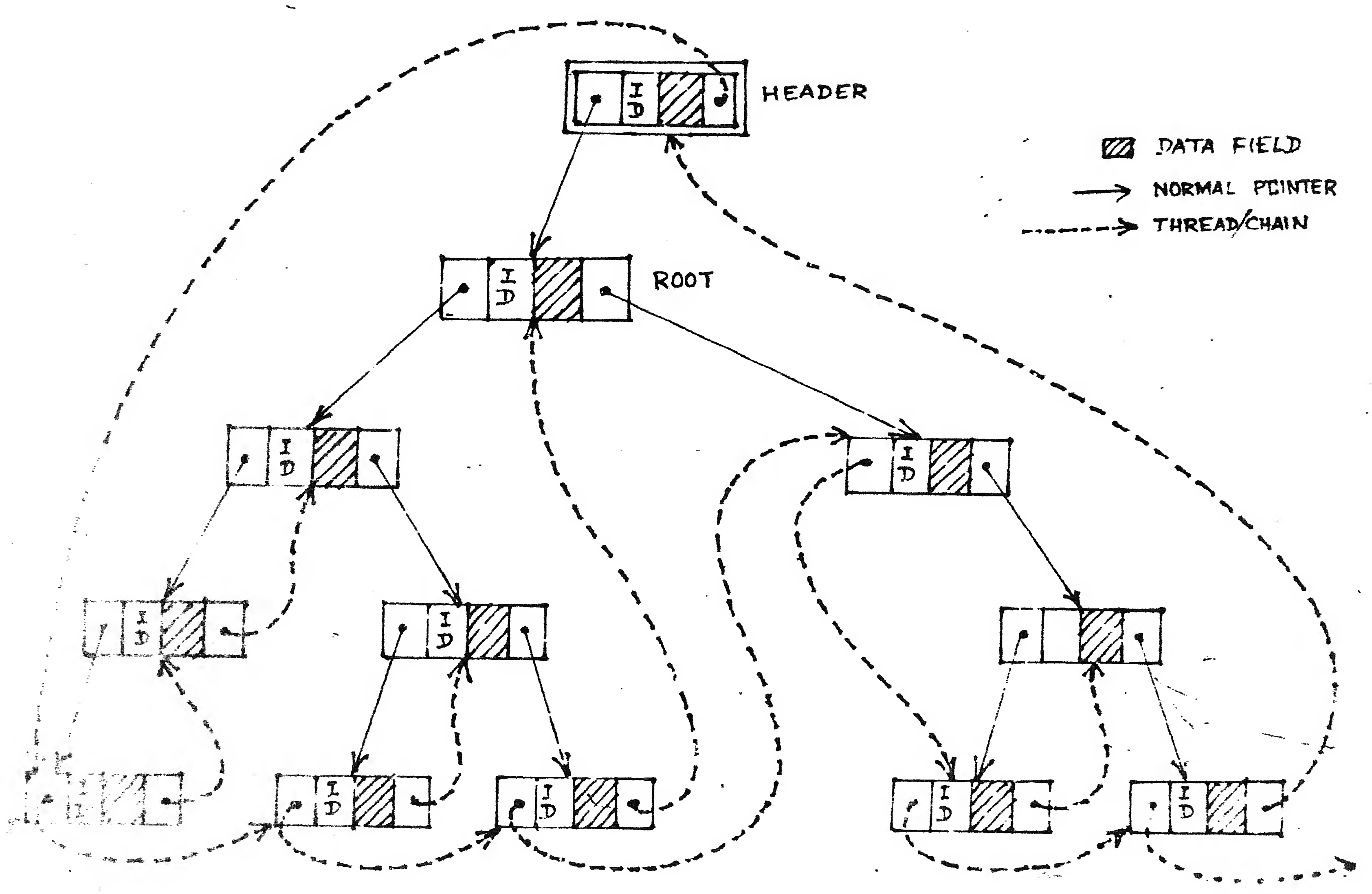


Fig. 2.5 A chained and threaded binary tree

The assertions characterising this structure follow :

Assertion 1 : $\forall$ N1 ε S, $N_2 \neq$ nil

(a) $f_1(N_1) = N_2 \Rightarrow N_2 \varepsilon S$

(b) $f_2(N_1) = N_2 \Rightarrow N_2 \varepsilon S$

This is identical to the previous case.

Assertion 2 :

(a) $\forall$ N1, N2, N ε S $(N_2 \neq N_H)$

$(t_1(N_1) = 0$ <u>and</u> $f_1(N_1) = N_2)$

or $(t_2(N_1) = 0$ <u>and</u> $f_2(N1) = N_2)$

$\Rightarrow$ $f_1(N) \neq N_2$ for any $N \neq N_1$ s.t. $t_1(N) = 0$

<u>and</u> $f_2(N) \neq N_2$ for any $N \neq N_1$ s.t.$t_2(N) = 0$

(b) $f_1(N) \neq N_H$ for any N ε S. s.t.$t_1(N) = 0$

<u>and</u> $f_2(N) \neq N_H$ for any N ε S s.t.$t_2(N) = 0$

This assertion just states that the 'normal pointers' do not form a cycle.

When the tag = 1, indicating that the associated field contains a thread or chain, we can look upon those fields to contain a value dictated by a functional associated with that field. These are single valued functionals on the nodes of the structure. Some typical examples of the functionals are :

F1 The Inorder successor function

F2 The function which gives the next node N_2 in the inorder enumeration sequence such that tag(N_2)= 1.

This is stated in the following assertion.

Assertion 3 : (a) $\forall_{N_1} \varepsilon\, S$ such that $t_1(N_1) = 1$

$$f_1(N_1) = F_1(N_1)$$

(b) $\forall_{N_1} \varepsilon\, S$ such that $t_2(N_1) = 1$

$$f_2(N_1) = F_2(N_1)$$

When the examples of F_1 and F_2 given above are fitted in the assertion, the structure becomes a chained and threaded binary tree.

Assertion 4 : (a) $f_1(N_1) = N_2 \Rightarrow I_d(N_1) = I_d(N_2)$

(b) $f_2(N_1) = N_2 \Rightarrow I_d(N_1) = I_d(N_2)$

This assertion assumes an identifier field to be associated with every node and containing a unique value throughout the structure.

2.4 The error model :

The types of errors which we consider for the purpose of this study, as also the assumptions made therein are presented below.

2.4.1 Assumptions : The Valid State Hypothesis

The basic assumption which we make in the model, which will also be carried over throughout the analysis is what is called the 'Valid State Hypothesis [7]. The hypothesis can be stated as under :

- Links outside a particular instance do not point to any node within the instance, and
- The unique identifier value of an instance, whenever used, appears only in its own identifier fields.

Thus, no node in the 'garbage' either contains a pointer to any node in the structure under consideration or contains an I_d-field that is unique to the structure.

The Valid State Hypothesis seems a little unrealistic at the first sight, as it demands something about the 'garbage' and other structure instances also. But once the Valid State Hypothesis is maintained true in a system, then it could continue to be maintained if all deletions of nodes (whenever they are released to the garbage) are accompanied by the process of making the pointer fields of deleted nodes nil and changing the I_d-field content to a value that is unique to the nodes in the garbage. This ensures the validity of the hypothesis. All data-accessing programs are supposed to abide by these norms.

2.4.2 The Errors

We allow an arbitrary change of any arbitrary field(s) of arbitrary node(s). Any hardware error or software error could be modelled as above. Unless otherwise stated, the analysis and synthesis would pertain to errors occurring not only in pointer fields but also in the Id-fields and count, wherever present. We, however, preclude errors occurring in the 'data fields' as our subject matter concerns only errors in structural information in a data structure. Multiple errors are unintentional changes in more than one field of a single node or in identical or different fields of different nodes.

An error in terms of the abstract mathematical model would be violation of one or more of the assertions describing that structure.

Also, it is possible, that an assertion could be violated by more than one type of error.

A note about the class of faults called 'Macro faults' by Black et al. [1] would be in order. These faults are those in which all the structural information contained in a node are lost and can be thought of as an arbitrary number of multiple faults (in fact, as many errors as there are fields in the node) occurring in that node. This class, therefore, is not separately treated.

3. DETECTABILITY

In this section, formal proofs about the detectability of some standard data structures are given in the light of the model proposed in Section 2. Each such proof is accompanied by an informal argument about the detectability. Most of the proofs are not just proofs of detectability but also tell which assertions are violated by which types of faults.

The analysis assumes the error model mentioned in Section 2. Only structural errors, like errors in stored count, Id-field and pointer fields, are therefore considered.

Consequently, for linked data structures, the errors could be classified into one or more of the following :

1) Error in Pointer fields
2) Error in stored count, if a stored count exists
3) Error in Id-field, if an Id-field exists.

Error is caused by unintentional change of any of the above mentioned fields.

In the following subsections a reference to assertion 1 etc. would refer to the indicated assertion for that type of data structure, as found in Section 2.

3.1 SLL :

A singly linked linear list with no redundancy has only one assertion, so no redundancy exists. We note here, that any additional assertion amounts to redundancy, as just assertion 1 is sufficient to access all the nodes of a linearly linked list.

Let the error occur in node N_1 belonging to S.

Let $f_1(N_1) = N_2$ previously before the change and let it change to $f_1(N_1) = N_3$. Now, by assertion 1 , $N_3 \varepsilon S$. There is no way to find out whether the change was intentional or an error.

Hence the detactability is zero.

3.2 Robust-SINGLY LINKED LIST (r-SLL) :

(This structure has an Id-field in each of the nodes in addition to a stored count of the number of nodes in it).

(a) Change in Id-field :

Let $N_2 \varepsilon S$ and $Id(N_2)$ be in error.

Then, $\exists N_1 \varepsilon S$, s.t. $f_1(N_1) = N_2$

(This follows from assertion 1)

But assertion 2(a) states that

$$Id(N_1) = Id(N_2)$$

An invalidity is reached here in assertion 2(a).

(b) Change in stored count :

Let the old count be m and the new value, after error, be n($\neq$ m).

(There is no other change, since we are now concentrating only on single errors).

Initially f_1^k $(N_H) = N_H$ only for $k = m+1$.

Hence, f_1^{m+1} $(N_H) = N_H$.

Also, f_1^{n+1} $(N_H) = N_H$, after the change.

Since n is different from m, $n+1 \neq m+1$.

Therefore, assertion (3) is violated.

(c) Pointer field change :

Let the pointer field of N_1 change.

Initially, let $f_1(N_1) = N_2$, later $f_1(N_1) = N_3$.

If $Id(N_3) \neq Id(N_1)$, then N_3 is a 'foreign' node and assertion 2(a) is violated.

If $Id(N_3) = Id(N_1)$, then, under the assumption that there is only a single change, N_3 belonged to the structure S previously too.

$$\text{Let } f_1^{k_1} (N_H) = N_1$$

$$f_1^{k_2}(N_3) = N_H \text{ for some } k_1, k_2 \geqslant 0$$

Also, we know that, $f_1(N_1) = N_3$.

Hence, $f_1^{k_1+k_2+1}(N_H) = N_H$

So, $k_1+k_2 = m$, if assertion (3) were valid.

There are now 2 cases :

Case 1 : $k_2 < m-k_1$

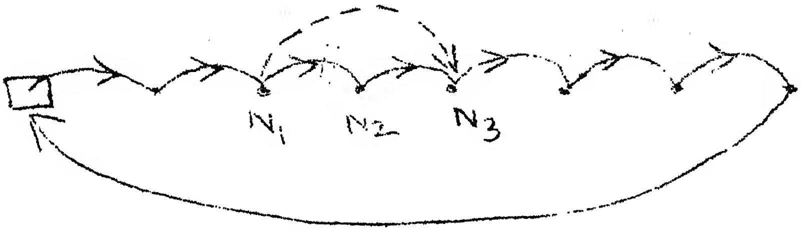


Fig. 3.1 Illustrating case (1) of the proof
$m=7$, $k_1=2$, $k_2=4$ $(k_2 < m-k_1)$

In this case,

$$k_1+k_2 = m \Rightarrow k_1+m-k_1 < m \text{ , or } m < m$$

A contradiction is reached here. Hence, assertion (3) could not be true in this case, and is the one that is violated.

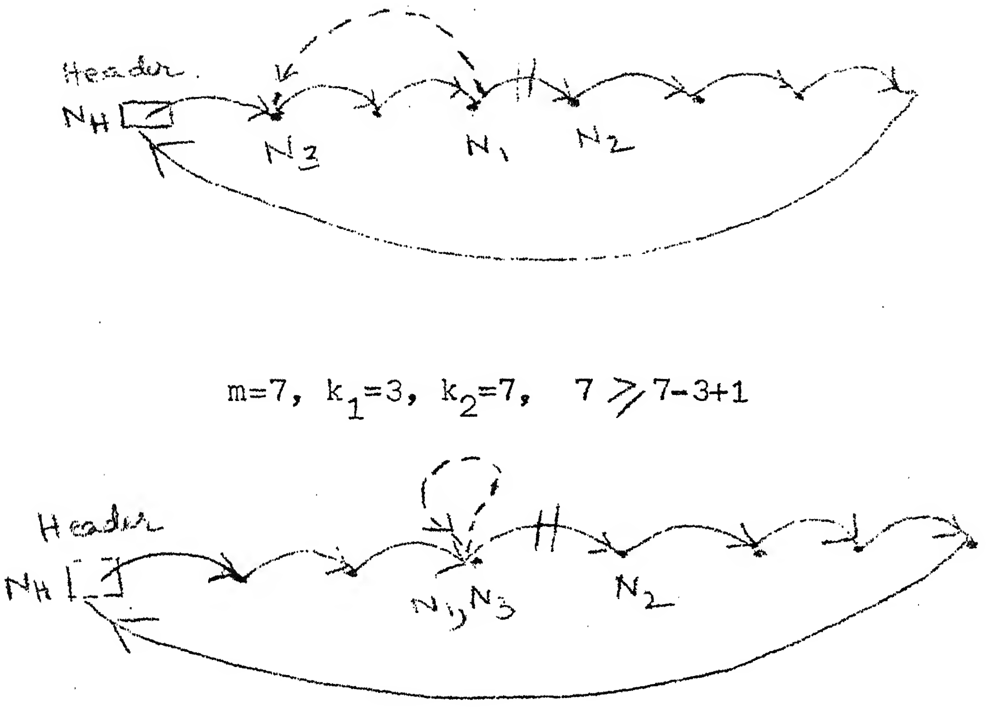
Case 2 : $k_2 \geqslant m-k_1+1$

(In this case, $k_1+k_2 = m \Rightarrow m+1 \geqslant m$ which is not much information).

Observe that $f_1^{k_1}(N_H) = N_1$

$$f_1(N_1) = N_3$$
$$f_1^{k_2}(N_3) = N_H$$

with $k_2 \geqslant m+1-k_1$.



$m=7,\ k_1=3,\ k_2=7,\quad 7 \geqslant 7-3+1$

$m=7,\ k_1=3,\ k_2=5,\quad 5 \geqslant 5$

Fig. 3.2 Illustrating the case (2) of the proof ($k_2 \geqslant m-k_1+1$)

It follows that $f_1^{k_3}(N_H) = N_3$ for some $k_3 \leqslant k_1$.

Hence,
$$\left.\begin{array}{l} f_1^{k_3}(N_H) = N_3 \\ \text{and} \\ f_1^{k_1}(N_H) = N_1 \end{array}\right\} \text{ for some } k_3 \leqslant k_1$$

Therefore, $f_1^{k_4}(N_3) = N_1$ for some $k_4 \geqslant 0$

So, $f_1^{k_4+1}(N_1) = N_1$, since $f_1(N_1) = N_3$.

It, then, follows that

$$f_1^m(N_H) = N_H \Rightarrow f_1^{m+k_4+1}(N_H) = N_H$$

The uniqueness of 'k' in assertion (3) is lost and so assertion (3) is violated.

Informal Arguments :

Assertions (2) and (3) are redundant as far as access of nodes are concerned. Together, they add to the detecting capability of the structure. It should be noted that, every 'redundant' assertion might not increase the detectability by one. As in this case, assertions (2) and (3) cater for different types of errors, and, together 'cover' all the errors, so, together, they increase the detectability by one. Assertion (2) helps in detecting a pointer change to a foreign node and assertion (3) will help in detecting some nodes getting deleted by a single pointer change.

We can summarise the above discussion by recapitulating the results. Mentioned below are the detecting capabilities of various assertions :

1) Id-field change : Assertion 2a detects
(No other procedure is able to detect)

2) Count change : Assertion 3 detects

3) Pointer field change :

a) To a foreign node : Both assertions 2a and 3 detect

b) To a local node : Only assertion 3 detects

3.3 DOUBLY LINKED LIST (DLL) :

The types of errors here are :

1) Error in Forward pointer (f_1-field)

2) Error in back pointer (f_2-field)

3) Error in stored count (m)

4) Error in Id-field (Id-field)

Any single error is 'detected' by violation of identical assertions as in the robust singly linked list (r-SLL) case.

In addition, the errors of type (1) and (2) are also detected by violation of assertion 2.

Any double error, not involving both pointer fields f_1 and f_2, is also detected likewise and we will not go into its details.

The only interesting case is when both the pointer fields, f_1 and f_2, of two nodes change.

Let $f_1(N_1) = N_2$ change to $f_1(N_1) = N_3$.
If the f_2 field of N_3 does not change, then this error is detected by violation of assertion 2.

So, assume $f_2(N_3) = N_4$, changes to $f_2(N_3) = N_1$.
Assertion (2) is now held valid, but assertion (4) is found to be violated, as in the case of robust singly linked lists.
If N_3 is a foreign node, then assertion 3 itself is violated.

Thus, all double errors are detected and so, the detectability is two.

An instance of 3 errors that goes undetected is a pair of pointer changes involving f_1 and f_2 fields together with a change in stored count. Informally, apart from the detection capability of r-SLL, the DLL has got an additional detectability because of assertion 2. Hence, the detectability is one more than that of r-SLL. Recapitulating, we can present the results of the above discussion by listing out which types of errors are detected by what assertions.

1. Id-field change : Assertions 3a and 3b detect
2. Count change : Assertions 4a and 4b detect
3. Forward pointer (f_1-field) change :

   a) To a Foreign node : Assertions 2a,2b,3a and 4a detect

   b) To local node : Assertions 2a,2b and 4a detect

4. Back pointer (f_2-field) change :

   a) To a foreign node : Assertions 2a,2b,3b and 4b detect

   b) To a local node : Assertions 2a,2b and 4b detect.

These results will be used later in Section 4 to construct the 'Syndrome Table'.

4. DETECTION AND RECOVERY PROCEDURES

Given a data structure described in terms of a set of assertions to be satisfied, a semi-automatic method of coming up with detection procedures for structural errors is now presented.

4.1 Procedural Specifications of Data Structures :

Each of the assertions is converted to an algorithm or procedure which checks for the validity of the assertion. Each of the procedures will return a true/false value for the error flag associated with it. Any instance of the structure is made to pass through the procedures which are algorithmic equivalents of the assertions describing that data structure. A success or failure is indicated in each case by the value of the error flag. A success would mean no detected error and the corresponding error flag will be returned as false.

Note that many types of errors could be detected by a single procedure. Some of the procedures return a suspected node, whose field is to be modified when correction is attempted.

In converting an assertion to a procedure, it may be necessary to do the following changes, which are not present in the assertions :

i) Some procedures also return a suspect node, as noted above, though the assertions do not say anything about it.

ii) To terminate the algorithm, in particular, to avoid indefinite looping in some cases, some modifications might be needed.

The failure or success of each procedure is calculated and the aggregate of these results is used to detect as well as pinpoint the type (and possibly, the location) of error that has actually occurred.

A general method of doing so is indicated below.

4.2 The Syndrome Table :

The identification of the various types of possible errors that can occur in the data structure is a key step in what follows. In case of multiple errors, all combinations of single error types are to be considered as possible.

Having identified the various types of errors, an analysis of the data structure in the light of the assertions describing it is to be carried out on the lines of Section 3. It will be able to tell which procedures return a true value for which type of faults (errors). This information, when presented in a tabular form is called the syndrome table.

The syndrome table for a robust singly linked linear list (r-SLL) with a stored count and Id-field is given below.

	Error flags	
	Error flag 2	Error flag 3
No error	F	F
Id-field change	T	F
Count change	F	T
FP(f_1) to foreign node	T	T
FP (f_1) to local node	F	T

Fig. 4.1 The syndrome table for SLL

The entries in the table show the values of the corresponding error flags upon exit from that procedure.

The syndrome table is a master table which will be used not only to detect errors but also to pinpoint the type of error and, coupled with the suspected node values, to locate the error in some cases. This aspect is important to correction. We are not only able to detect, but also diagnose the errors.

A row in the table is a 'syndrome' for that particular type of error. Any actual syndrome (a sequence of test results) obtained as a result of passing the instance of data structure through the various procedures is compared with the table to identify the type of error that could have led to the syndrome.

A note about the utility of the syndrome table will be in order. A particular syndrome table will be able to diagnose an error to be of a particular type only if the syndrome for that type is unique in the entire table. Otherwise, the resolution will be affected and it will only be possible to identify the actual error to be one among a set of types of errors and additional checks (perhaps, involving the suspect-node values) may be necessary to resolve further.

The procedures for detection for some of the standard data structures can be derived, and are presented by way of examples, along with the syndrome tables. The results of the analysis, given in Section 3, are utilised to construct the syndrome tables in each of the cases.

4.3 The robust singly linked list (r-SLL) :

We consider here, the robust singly linked list with a stored count and having an Id-field in each node. Its detectability is one.

4.3.1 Procedural specifications :

There are two 'redundant' assertions in the specification of an r-SLL (Section 2, page 11). These are assertions 2 and 3. Given below are their procedure equivalents.

Assertion 2 :

The procedure given below checks for the validity of assertion 2 of r-SLL over the entire node space of a given instance of data structure. The procedure checks for the assertion in a loop, each time checking for a different node. Termination of the procedure in a finite time is ensured by including the presence of an detected error or the number of nodes encountered so far exceeding the stored count as a termination condition for the loop.

```
Procedure 2 (NH : Node);
   {checks for the validity of assertion 2 of r-SLL}
    Global Var Error 2: Boolean;
               Suspect 1, Suspect 2: Node;
    Var        $N_1$,$N_2$: Node;
               m, nodecount: Integer;
    Begin
       N1:= $N_H$; Error 2 := false; Suspect 1: = nil;
                                      Suspect 2: = nil;
```

contd ...

```
{ Begin
    N1:=$N_H$; Error 2 :=false; Suspect 1:=nil;
                                    Suspect 2:=nil; }
    nodecount: = 0; m:=count ($N_H$);
    repeat
        N2:=$f_1$($N_1$);Nodecount:=nodecount +1;
        If(Id(N1) = Id($N_2$))
            then N1:=N2
            else Error 2:= True
        until (N2=$N_H$) or (error 2) or (nodecount $\geq$ m+1)
        If error 2
            then begin
                Suspect 1: = $N_1$
                Suspect 2: = $N_2$
            end
  end;
```

Assertion 3 :

The procedure given below checks for the validity of the assertion 3 of the specifications of r-SLL given in Section 2 (page 11) over all the nodes in an instance. It checks if the stored count matches with the counted number of nodes when a traversal is made.

```
Procedure 3 ($N_H$ : Node);
    {checks for validity of assertion 3 of r-SLL}
     Global Var Error 3: Boolean;
     Var      N1,N2: Node; m, nodecount: Integer,
     Begin
        N1:=NH; K:=O; Error 3:=false; m:=count($N_H$);
        repeat
           N2:=$f_1$(N1); k:=k+1;
           If (N2=NH)
              then error 3:= $k \neq m+1$
     until (k = m+1) or (error 3);
     If (k = m+1) then error 3:=$N2 \neq NH$
  end;
```

4.3.2 The syndrome table for r-SLL (single errors) :

There are basically three types of errors :

1) Id-field change (Id)

2) Count change (m)

3) Forward pointer change (f_1)

The last can be further split up into two :

a) Forward pointer changing to a 'foreign' node

b) Forward pointer changing to a 'local' node

These are exactly the same types of errors that were considered in Section 3 for analysis purposes. Use is made of the summarised results presented in Section 3 (page 26).

Only point to note is that if an assertion is used to detect a particular type of error, the assertion is going to return a true value for the error flag associated with that assertion. For example, since an Id-field change is 'detected' by assertion 2, we can say that, in the presence of this error, erro2 will be true and error 3 will be false.

The syndrome table, therefore, looks as shown in Fig. 4.1.

4.3.3 Comments :

It is observed from the syndrome table that any erroneous instance produces a syndrome that is different from the one for the correct one, so detection is guaranteed. Since two of the rows are identical, it is not possible to differentiate between a change in stored count and a pointer changing to a node in the same instance. This is expected, as it is an inherent property of the r-SLL that its correctability is zero. But, it can also be observed that not all single errors can be diagnosed, though in some cases, it is possible. (Diagnosis would mean finding the type of error and the location where it has occurred).

4.4 The Doubly Linked list (DLL) :

The doubly linked list, whose assertions are given in Section 2 (page 12) has a detectability of two. Construction and comments about its syndrome table follow.

4.4.1 Procedural specification :

The procedural equivalents of the assertions that specify a DLL are given in Appendix 3.

4.4.2 The syndrome table for DLL (The single error) :

The following types of faults are identified :

1) Change in Id-field (Id)

2) Change in stored count (m)

3) Change in forward pointer (f_1)

4) Change in back pointer (f_2)

Again, the types of faults (3) and (4) can be subdivided into the following classes :

a) The pointer changing to a 'foreign' node

b) The pointer changing to a 'local' node.

When only one of the above types of faults is assumed to be present, it is a case of single errors in a doubly linked list. Here, it can be observed that the errortype resolution (diagnosis) is perfect in case of single errors, a prerequisite for correction, as expected.

The above are exactly the types of errors considered in Section 3 for the analysis of DLL (page 27). The summary of the results presented therein is used to construct the syndrome table.

The syndrome table, therefore, is as shown in Fig. 4.2.

Error flags

	Error 2a	Error 2b	Error 3a	Error 3b	Error 4a	Error 4b
No error	F	F	F	F	F	F
Id-field change	F	F	T	T	F	F
Count change	F	F	F	F	T	T
f_1 change to foreign node	T	T	T	F	T	F
f_1 change to local node	T	T	F	F	T	F
f_2 change to foreign node	T	T	F	T	F	T
f_2 change to local node	T	T	F	F	F	T

(Fig. 4.2 The syndrome table for DLL)

4.4.3 Comments :

Since the correct instance results in a syndrome that is different from the syndromes for any of the erroneous instances, detectability is guaranteed for single errors. Also, since no two rows are identical, the diagnosis is perfect : given a syndrome arising out of a set of tests, we can pinpoint the type of error. Then, in conjunction with the knowledge about the suspect-nodes, the location of the error can also be identified.

In fact, some of the test procedures like 4a and 4b could be avoided if use is made of the suspect-nodes returned by other procedures in diagnosis. For example, in the absence of procedures 4a and 4b, it appears from the syndrome table that a forward pointer change to a local node is indistinguishable from a back pointer change to a local node. However, this can be resolved by looking at the suspects returned by each procedure. The method is indicated in Appendix 1.

4.5 The sequence of application of test procedures :

Though the procedures testing the validity of the assertions are given independent of each other, and can be used as such, it may be more efficient to apply them in a predetermined sequence.

Some of the types of errors are uniquely identified even by a few of the test results (a subsyndrome) and in such cases, it is not necessary to find out the full syndrome. Those test procedures which suffice to uniquely identify a syndrome belonging to the syndrome table are to be applied prior to the application of other test procedures to avoid application of unnecessary procedures.

An example, in case of DLL is to first apply 2a and then 3a so that an Id-field change, if present, is detected without the application of other redundant procedures (Hence, on this count, a good sequence will first apply procedures 2a and 3a before other procedures).

Another reason why not all the test procedures need to be applied is exemplified by the correction algorithm for a doubly linked list given in [6] and reproduced in Appendix 2. In effect, it uses only test procedures 2a and 2b but makes use of the knowledge about suspects returned by each of these procedures. It is seen that this information is sufficient to correct all pointer errors. Hence, if we decide to use the information about the suspects in diagnosis and correction, it is better to apply procedures 2a and 2b before application of other procedures, so as to detect and correct any pointer error that is present. None of the other procedures is able to cover such a broad class of errors.

4.6 Correction of Data structures :

4.6.1 Introduction :

Having diagnosed an error to be of a particular type, we now consider some general remarks about the correction of those errors. Taylor and Black have given a set of five general principles which a data structure and its correction algorithm should satisfy for correction [8]. Data partitioning, avoiding unbounded foreign travel, coroutine traversal of data structures, using a fault dictionary and guessing in case of doubts are the five principles enunciated in it. It is to be noted that correction is a much more difficult problem than detection or diagnosis and the development of a correction procedure may not be straightforward.

A general way to correct erroneous data structures for single errors, knowing the type and location of the error, is now presented. The assertions are again assumed to be represented by procedures which check for their validity. Within the procedure, there will be a particular condition which will be checked to detect a violation of the assertion, and the identification of that condition is the key to correction. Mostly, these conditions are in the form of some equality. Its violation is an error. The correction procedure, therefore, attempts to set this equality right. The correction is achieved by changing the appropriate field of the culprit node to re-establish the equality.

For example, if error 3a in a DLL was returned as true, it means Id(N1) and Id(N2) were not equal where, N1 is suspect 1_{3a} and N2 is suspect 2_{3a}. In combination with other procedures, it is possible to identify the culprit among the suspects as that is what precisely constitutes diagnosis. Let us assume that the culprit node is suspect 2_{3a} and the error type is diagnosed to be a change in Id-field. Then, all that is needed for correction is to make Id(culprit) = Id(other suspect).

4.6.2 Correction procedures :

The Diagnosis-cum-correction procedures for the two data structures, r-SLL and DLL are now presented.

(a) robust Singly Linked List :

It should be noted that the 'diagnostability' and correctability are not one, so correction of single errors may not always be possible.

The procedure given below follows directly from the syndrome table for r-SLL and the detection procedures.

```
If (error 2)
   then  If (not error 3)
         then begin
              culprit := suspect $2_2$ ;
              Id(culprit):=Id(suspect $1_2$)
         else
              culprit:=suspect $1_2$;
              'uncorrectable error'
         else 'undiagnosable  error' ;
```

(b) Doubly linked list (DLL) :

The following routine may be used for correction of single errors in a doubly linked list. The syndrome table is used in diagnosis and the detection procedures help in finding the equality condition which must be re-established for correction.

```
If (error 3a) and (not error 2a)
   then
      begin
         culprit:=suspect $1_{3a}$;
         Id(culprit):=Id(suspect $2_{3a}$);
         return
      end;
If (error 3b) and (not error 2b)
   then begin
         culprit:=suspect $1_{3b}$;
         Id(culprit):=Id(suspect $2_{3b}$);
         return
   end;
Ir (error 2a)
   then If suspect $1_{2a}$=suspect $2_{2b}$
        then begin
              culprit:=suspect $1_{2a}$;
              $f_1$(culprit):=suspect $1_{2b}$
              return
         end;
If (error 2b)
   then begin
         culprit:=suspect $1_{2b}$;
         $f_2$(culprit):=suspect $1_{2a}$;
         return
   end;
```

```
If (error 4a) or (error 4b)
   then begin
        count $(N_H)$:=K
        {K is the nodecount}
   end;
```

Note the similarity between this procedure and the one presented by Taylor et al. in [6], which is also reproduced in Appendix 2. The above procedure, however, is derived entirely from the syndrome table and the detection procedures for DLL in a systematic manner.

4.7 On-line Error Treatment :

On-line error detection refers to detecting an error at the time when a particular portion of the data is being accessed. A similar notion applies to online error correction. We refer to the detection/diagnosis/correction problem as error treatment. Online error treatment is an advantage in cases where the nature of the transactions on the data is such that it is not possible to halt the system for purposes of error-checking. In a database environment, the need to take a 'before-image' is eliminated. Also, if the data is critical and the errors could not be tolerated, then, it is not advisable to do only a periodic run of the error detection routines, as the data between two such runs might have been in error, as also all the transactions done on such data. Under such circumstances, online error treatment scores over the periodic treatment method.

On -line error detection (correction routines can be easily incorporated into any package of access mechanisms for a data structure, provided the structure can be described in terms of assertions such that none of them need an elaborate search through the node space of the instance to verify the assertion.

If such on-line error treatment routines are available, they could be included in the access mechanisms and the user need not be aware of it. Any access to data goes through these error treatment routines and an appropriate action is taken whenever an error is detected. If the error cannot be corrected on-line, the user may be given appropriate message. If it can be corrected, then it is done so, and the entire operations remain transparent to the user. Of course, all these, at the cost of efficiency of access, as every time the data has to go through the error treatment routines.

Such on-line error correction capability with complete transparency for single errors is possible in a doubly linked list (DLL) (with a stored count in the header and an Id-field in each node). Here, though procedures 4a and 4b require elaborate traversing of the entire data structure instance, the count field, by itself does not contribute anything to the redundancy power of the structure and it is only the count that the procedures 4a and 4b check for.

5. SYNTHESIS OF ROBUST DATA STRUCTURES

5.1 Introduction :

The synthesis of robust data structures is a difficult problem due to the following reasons :

a) The lack of proper design criteria to be satisfied by the resultant structure

b) The absence of design tools and methods to work with.

Though the required criterion could be stated to be detectability or correctability of some specified value, it is still a loose criterion. We shall later consider the number of changes required to maintain a valid instance with a different number of nodes (ch-diff) as a criterion [7].

5.2 Redundancy in a broader perspective :

Analysis and synthesis of robust data structures could be done under two seemingly different aspects.

As far as the analysis is concerned, we could study the effect of additional redundant fields, or alternatively, given a fixed number of redundant fields, the effect of using them in various ways could be studied.

An example illustrating these two aspects is the follows [6] :

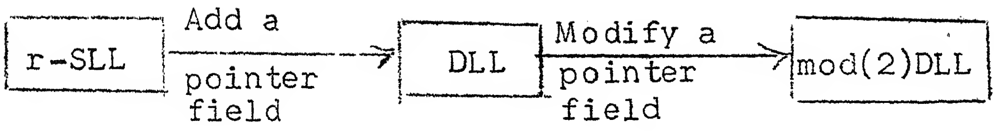


Fig. 5.1

The detectability of SLL goes from 1 to 2 by adding an additional field and resulting in the doubly linked list in the standard form. However, it is possible to increase the detectability to 3, not by adding an additional field, but by just using the available redundant fields in a better way. In a mod (2) DLL, the second pointer field points, not to the immediately preceeding node, but to the node preceding that.

On the synthesis side, these two approaches would mean improving 'robustness' either by adding redundant fields or by a better use of available redundant fields.

The second approach, i.e. efficient use of available fields, is considered now.

5.3 The redundancy functional :

The concept of redundancy functional (RF) was introduced in Section 2 (page 18) in connection with the 'F' function in chained/threaded binary trees. Every redundant field is looked upon as containing an information which is the value dictated by the redundancy functional for that field with that node as the argument. There may be different RFs associated with different redundant fields.

An advantage in using RF is that the synthesis problem which we now concentrate upon, is simplified. All structures with a given amount of redundant fields per node are characterised by identical assertions in their model except that their RFs differ. We can obtain different data structures by changing the redundancy functional..

In a doubly linked list, the assertion 2, viz.,

$$f_2(N_1) = N_2 \Rightarrow f_1(N_2) = N_1$$

can be restated as

$$f_2(N_1) \Leftrightarrow F(N_1)$$

where F is the redundancy functional.

If we choose F as the 'predecessor function', we get the standard DLL and if we choose F to be 'the predecessor of predecessor function' we get mod (2) DLL.

In the case of threaded trees [4], the redundancy functionals would be the 'In-order successor' (for the right link fields) and the 'Inorder predecessor' (for the left link fields) . For a chained tree [8], the RF is 'the next node in the Inorder sequence whose left (f_1) is nil',

In some cases, the RF can not be described by a single statement. Procedural definitions of RF may be needed in such cases. (The inorder successor function in case of threaded tree is an example of this kind).

5.4 mod(2) CT-tree :

We use the idea of redundancy functional to modify the chained and threaded binary tree (CT-binary tree [6]). It will be on similar lines as the modification of a standard doubly linked list into a mod(2) DLL.

5.4.1 Description of mod(2) CT-tree :

The mod(2) CT-tree is a binary tree in which all the right-link fields with a logically nil value are replaced by a 'thread' and all the left link fields with a logically nil value are replaced by a 'chain'. (Tag bits are associated with each pointer field to indicate whether the pointer is a normal pointer or a thread/chain. The pointer and the associated tag bit are assumed to be packed in a single 'field'.

Threads and chains are formed for a mod(2) CT-tree in the following way. Informally, a thread from a node will point to a node which was pointed to by the succeeding thread (in the inorder sequence) for a CT-tree implementation of

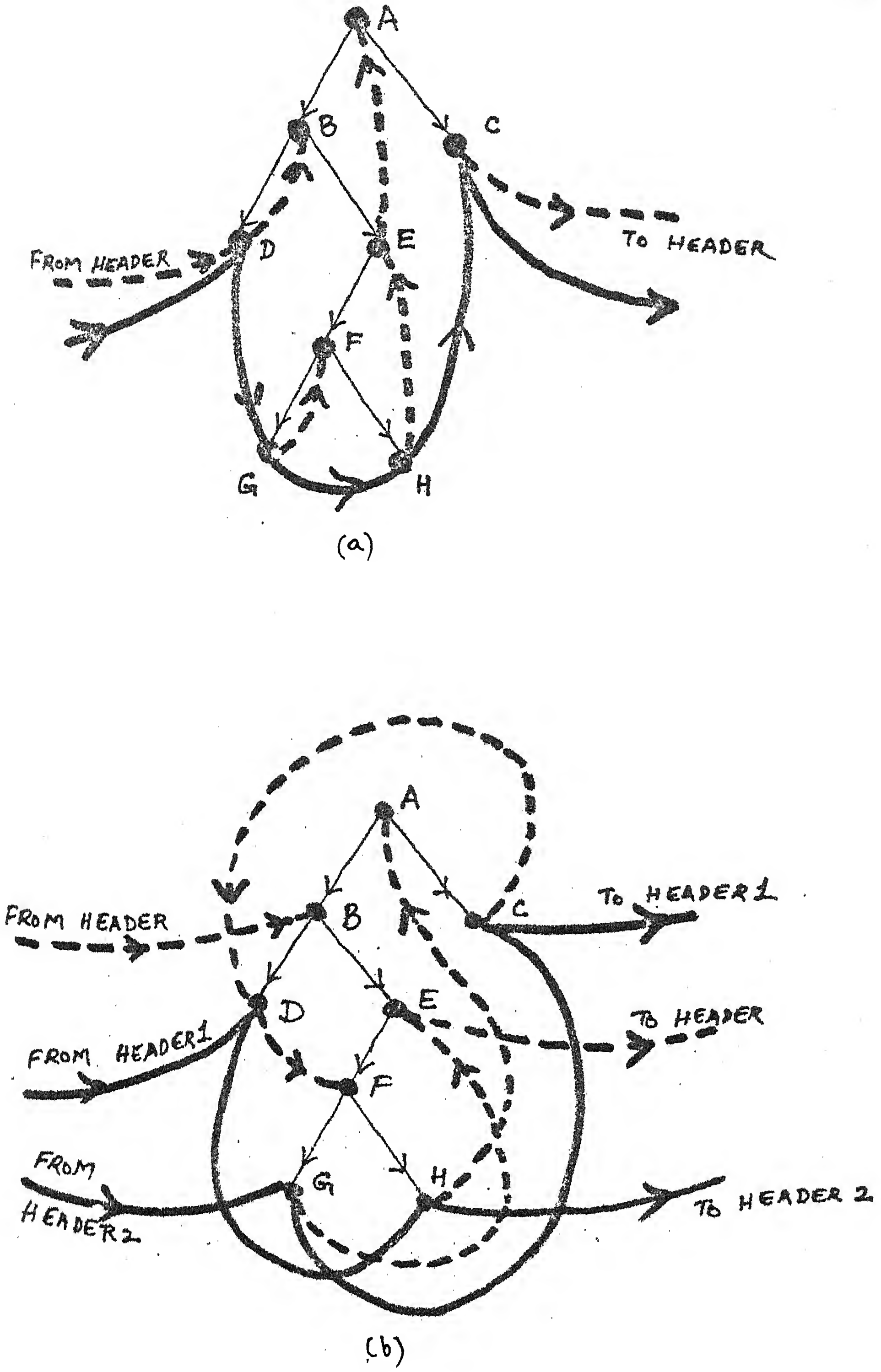


Fig. 5.2 (a) A CT-tree
(b) A Mod (2) CT-tree

the same data. Similarly, a chain from a node will point to a node which was pointed to by the succeeding chain in the corresponding CT-tree. (Refer to Fig. 5.2).

The redundancy functionals for the mod (2) CT-tree can be stated as follow :

Let N_1 be the node under consideration. Then, for threads :

> It returns a value which is the pointer to a node N_3 in the inorder successor of the node N_2 in the tree which is the next node in the inorder sequence after N_1 with a logical nil value for its right field,

and for chains :

> It returns a value which is the pointer to a node N_3 such that N_3 is the successor of successor of N_1, where 'successor' denotes that node which appears next in the inorder sequence with a logically null left field value.

5.4.2 Detectability of a mod (2) CT-tree :

The detectability of a mod (2) CT-tree is 3. This is shown by concentrating on ch-diff, the minimum number of changes required to convert a valid instance of the data structure to another valid instance with a different number of nodes. This essentially involves attempted deletion of some of the nodes and has the least lower bound compared to ch-same and ch-repl [7]. So, one less than ch-diff is the detectability.

The mod (2) CT-tree can be modelled using axiomatic assertions and can be analysed for its detectability. Given below is a different line of argument to show that its detectability is (at least) 3.

It is meaningful to talk of an attempted deletion of only a subtree in a tree structure. The subtree 'deleted' may be classified under one of the following :

Case 1 The subtree has root with degree zero. (In effect, it has got just one leaf

Case 1.1 The node A in the subtree is the leftson of its parent (Fig. 5.3a).

In this case, there is one chain and one normal pointer coming 'into' the node A, which have to be changed. In addition, there always exists a thread of the type from B to C, which has to be modified to point to D.

Case 1.2 The node A is the right son of its parent.

In this case also, there is one chain and one normal pointer coming into the node which have to be changed.

In addition, a chain of the type from P to C has to change to point to D. It is to be noted again that three pointer changes are required.

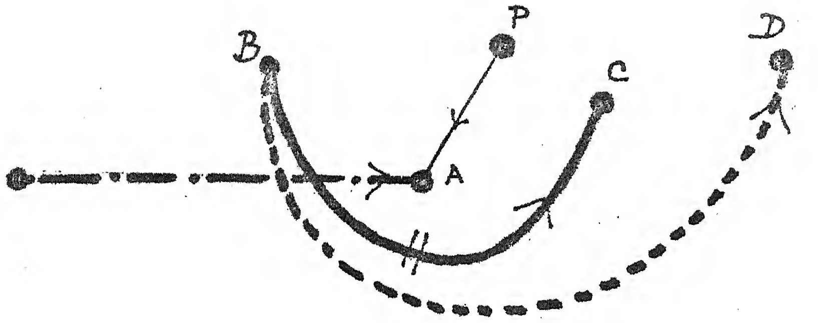


Fig. 5.3(a) Illustrating case 1.1
A is the node deleted
P is A's parent
B is the node with a null right field
and preceding A
C is the node with a null right field
and succeeding A
D is the inorder successor of C

⟶ NORMAL POINTER
➡ THREAD
-·→ CHAIN

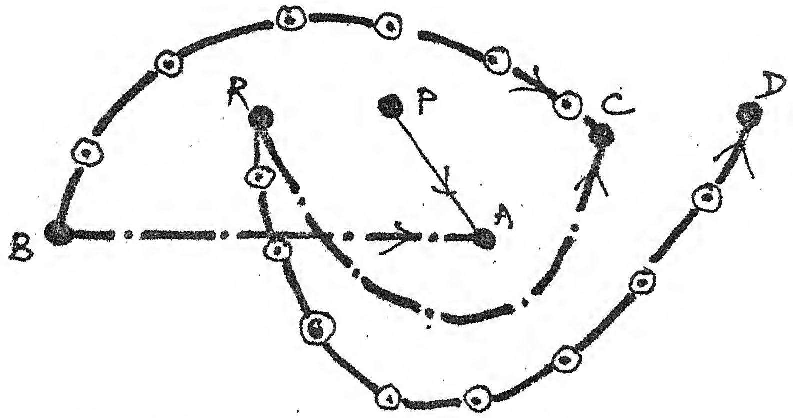


Fig 5.3(b) Illustrating case 1.2
A is the deleted node
P is A's parent
R is the node with a null left field
and preceding A
B is the node with a null left field & preceding **R**
C is the node which has a null left field
and succeeding A
D is the node with a null left field & succeeding C

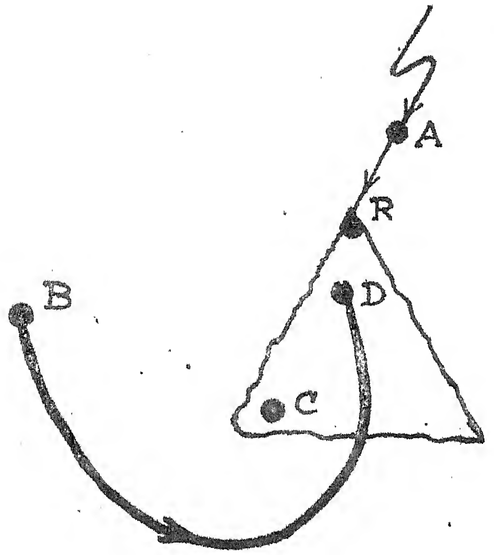
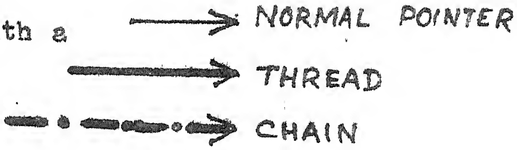


Fig. 5.4(a) Illustrating case 2.1
A is the root of subtree deleted
B is the node preceding C and with a null right field
C is the first leaf node in the subtree
D is the inorder successor of C



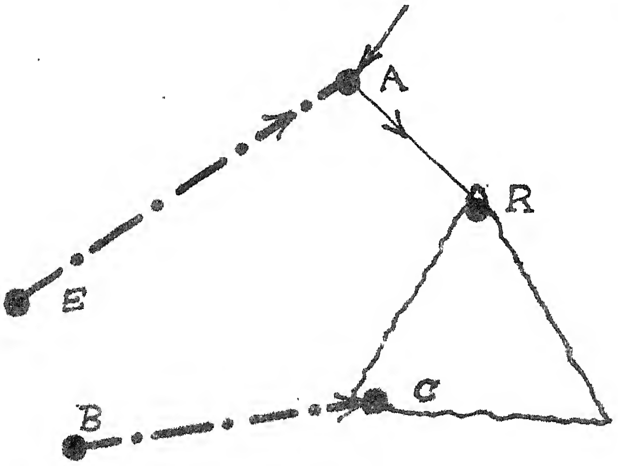


Fig. 5.4(b) Illustrating case 2.2
A is the root of subtree deleted
C is the first leaf node in the subtree
B is the node preceeding A and having a null left field

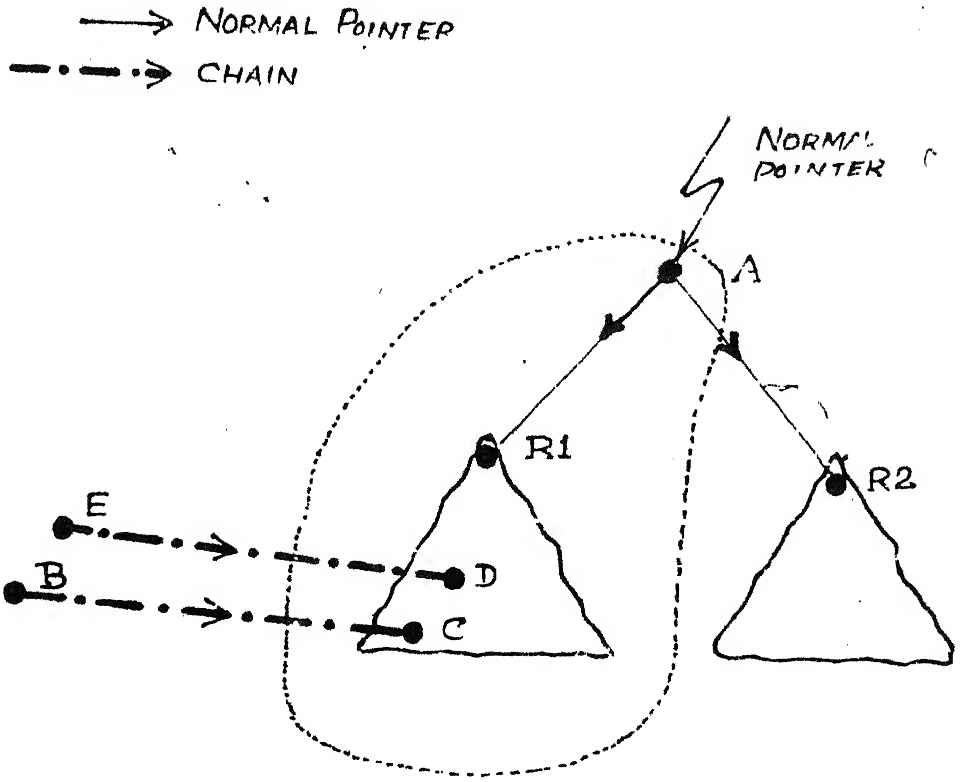


Fig. 5.5 Illustrating case 3
A is the root of the subtree, R_1 & R_2 are the two sons of A
C is the 'first' leaf of subtree A
D is the node with null left field and succeeding C
E is the node with null left field and preceeding C
B is the node with null left field and preceeding E

Case 2 The root is of degree 1.

Case 2.1 The root has a left son (Fig. 5.4a).

In this case, in addition to a normal pointer and a chain, there will be one thread (like BD in Fig. 5.4a), coming into the subtree. This thread, in fact, will point to the inorder successor of the first leaf of the subtree. These three pointers need to be changed when the subtree gets deleted. (The presence of an additional incoming chain is not guaranteed).

Case 2.2 The root has a right son (Fig. 5.4b).

In this case, in addition to a normal pointer and an incoming chain, which have to be changed, there will be another incoming chain (like BC in Fig. 5.4b), since there are atleast two nodes with logically null left fields in the subtree. (The presence of an incoming thread is not guaranteed) observe that, again, atleast three pointer changes are required.

Case 3 The root is of degree 2 (Fig. 5.5).

An attempt to delete such a subtree will involve changing atleast 2 incoming **chains** (which will be present anyway) and an incoming normal pointer, requiring atleast three changes.

If one of the subtree of the root, and the root itself are deleted, and the parent of the root A is made to point to the root of the other subtree, then, it will involve changing one normal pointer say to R2, atleast one incoming chain and either another incoming chain or an incoming thread, depending on whether the right subtree or the left subtree gets deleted along with the root.

In addition, in all of the above cases, the stored count has to change, requiring a minimum of 4 changes. Hence, the detectability is atleast 3.

5.5 Concluding remarks

It is observed that, by varying the redundancy functional, we can get variations of a data structure. This approach was used for synthesis using a fixed amount of redundancy. A storage structure, mod (2) CT-tree was developed, on the lines of mod (2) DLL and CT-tree, and was shown to have a detectability of (atleast) three. It is expected that a mod (k) CT-tree can be constructed on similar lines and the detectability will increase with increasing k (like mod (k) DLL) until the limiting factor (when ch-repl is exceeded by ch-diff) sets in. In case of chained and threaded trees it is expected that the limiting detectability will be 4, because ch-repl is 5.

6. CONCLUSION

Protecting data by adding redundancy in the structure seems to be a good idea, mainly because, it also aids in simplifying some user operations on the data. Such techniques can be used wherever large and critical data are being stored, like data bases. Though we have considered only pointer-type structures, it is justified by the wide use of such structures in many applications.

The detection and correction procedures presented in Section 4 are certainly not the most efficient. Nevertheless, they were derived in a systematic manner from the descriptions of data structures and this method will be useful particularly when the structure is complex.

It is felt that most of the problems of efficient use of available redundant fields can be dealt with using the redundancy functionals. The idea which was used to develop mod (2) CT-tree can be extended to other structures also to make them robust. Finding a proper design criterion to be satisfied by the redundancy functional is another direction in which this work can be extended.

The detectability and correctability presented here are worst case values. As per the results of simulation studies done by Taylor et al. [6], the data structures are found to behave much better than the predicted values, mainly because the chances of the worst combination of errors occurring is very small. This gap between theory and practice can be bridged if the probabilities of various types of errors occurring are also considered.

In general, the performance of the system goes down due to the presence of redundancy and more complex update routines. Thus, there is a trade-off between reliability and efficiency.

APPENDIX 1

Distinguishing between a forward pointer change to a local node and a back pointer change to a local node in DLL

In these cases, the following are the values of error flags :

Error 2a = True

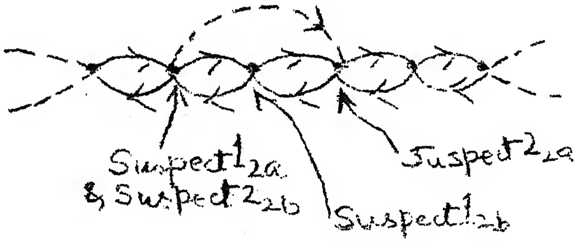
Error2b = True

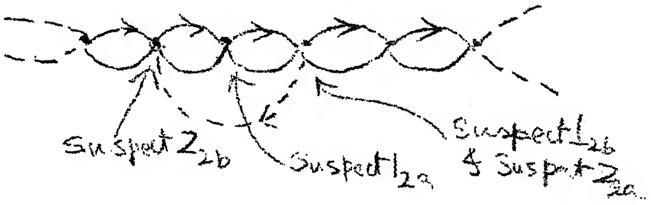
Error3a = False

Error3b = False

We do not resort to procedure 4a or 4b for this diagnosis.

Each of the procedures 2a, 2b, 3a and 3b (given in Appendix 3) return two suspects, suspect 1 and suspect 2. The following procedure will distinguish between the two types of errors in question.





```
If suspect $1_{2a}$ = suspect $2_{2b}$
    then errortype:='forward pointer changing to a local
                                        node'
    else If suspect $1_{2b}$ = suspect $2_{2a}$
        then errortype:= 'back pointer change to local node'
        else 'multiple errors';
```

APPENDIX 2

Reproduction of a linear list correction procedure from [6]

The following procedures detect and correct all single errors in a DLL.

```
Procedure LIST-CORR (H,N);
  begin
     pointer H, Integer N, Pointer P,
     Pointer Prev-P, Integer J;
     /*H is the header of the structure, N is expected
                          count */
     J ← 0;
     Prev-P ← H;
     P ← Forward (H);
     While (P ≠ H) do
        begin
           J ← J+1;
           If (Back(P)= Prev-P)  and (Id(P) correct)
               then begin
                    Prev-P ← P;
                    P ← Forward (P)
               end
            else
               begin
```

```
             {else
                   begin }
                       BACKSCAN (H,P,Prev-P);
                       return
                   end
            end;
       If (Back(H)≠Prev-P) or (Id(H) incorrect)
            then begin
                       BACKSCAN(H,P,Prev-P)
                       return
                  end;
         If(J≠N) then N ← J
end;
Procedure BACKSCAN (H,P,Prev-P);
       begin
           Pointer H, Pointer P, Pointer Prev-P,
           Pointer Q, Pointer Prev-Q;
           Prev-Q ← H;
           Q ← Back (H);
           repeat
               begin
                   If (Forward(Q)=Prev-Q)and(Id(Q)correct)
                       then begin
                           Prev-Q ← Q;
                           Q ← Back(Q)
                       end
```

```
        {repeat
           begin
             If (Forward(Q)=Prev-Q)and(Id(Q)correct)
                then begin
                     Prev-Q ← Q;
                     Q ← Back(Q)
                end}
             else
                begin
                    L-REPAIR(P,Prev-P,Q,Prev-Q);
                    return
                end
            end
        end;
Procedure L-REPAIR($P_1$ Prev-P,Q,Prev=Q);
    begin
      Pointer P,pointer Prev-P,pointer Q,Pointer Prev-Q;
      If (P=Q and Id(P) in-correct)
         then Id(P) ← correct i.d.
         else If (P=Prev-Q)
              then Back (Prev-Q) ← Prev-P
              else If(Prev-P=Q)
                   then forward (Prev-P) ← Prev-Q
                   else 'multiple error'
```

APPENDIX 3

Given below are the procedural equivalents of the assertions used to describe a DLL in Section 2.

Assertion 2 :

These assertions check whether the forward and back pointer of 'adjacent' nodes are properly formed. In procedure 2a, checking assertion 2a, the traversal is done using forward pointers, while a check on the previous node encountered is made. Procedure 2b uses traversal via back pointer (f_2). Either of the procedures terminate when an error is detected or if the header is reached.

```
Procedure 2a ($N_H$: node);
   Global Var Error 2a: Boolean; suspect $1_{2a}$,
                                         suspect $2_{2a}$: Node;
   Var       $N_1$,$N_2$: Node;
   Begin
      N1:=$N_H$; Error 2a:=false; suspect $1_{2a}$:=nil;
                                   suspect $2_{2a}$:=nil;
      repeat
           N2:=$f_1(N_1)$;
           If ($f_2(N_2)$=$N_1$)
              then N1:=N2
              else error2a:=true
           until(N2=$N_H$) or (error2a);
           If error 2a then
```

```
        { until (N2=$N_H$) or (error2a);
          If error2a then }
              begin
                 suspect $1_{2a}$:=N1;
                 suspect $2_{2a}$:=N2
              end
     end;
Procedure 2b($N_H$:Node);
     Global Var Error2b: Boolean;
                 suspect $1_{2b}$, suspect $2_{2b}$:Node;
     Var
                 N1,N2:=Node;
     Begin
       N1:=$N_H$; error2b:=false;
       suspect $1_{2b}$:=nil; suspect $2_{2b}$:=nil;
       repeat
                 N2:=$f_2$(N1);
                 If ($f_1$($N_2$)=N1)
                    then N1:=N2
                    else error2b:=true
       until(N2=$N_H$) or (error2b);
       If error2b
          then begin
                  suspect $1_{2b}$:=N1;
                  suspect $2_{2b}$:=N2
               end
       end;
```

Assertion 3 :

The following procedures check for the validity of assertions 3a and 3b. It is checked whether adjacent nodes have same Id-values or not. Procedure 3a uses traversal along forward pointer and procedure 3b uses the back pri pointer for traversal. A count of the number of nodes is kept in both cases to ensure termination of the procedure within a finite time.

```
Procedure 3a($N_H$):Node);
    Global Var error3a: Boolean;
            suspect $1_{3a}$,suspect $2_{3a}$: Node;
    Var
            N1,N2: Node; m, nodecount: Integer;
     {checks for validity of assertion 3a of DLL}
    Begin
            N1:=NH; error3a:=false; suspect $1_{3a}$:=nil;
            suspect $2_{3a}$:=nil; nodecount:=0;
            m:=count ($N_H$);
          repeat
                N2:=$f_1$($N_1$);Nodecount:=nodecount+1;
                If (Id(N1)=Id(N2))
                    then N1:=N2
                    else error3a:=true
          { then N1:=N2
            else error3a:=true}
        until (N2=$N_H$) or (error3a) or (nodecount $\geq$ m+1);
        If error3a then
            begin suspect $1_{3a}$:=N1;
                  suspect  $2_{3a}$:=N2;
            end
end;

Procedure 3b($N_H$:Node);
    Global Var error3b:Boolean;suspect $1_{3b}$,
                                    suspect $2_{3b}$:Node;
    Var         N1,N2:Node; m,nodecount:Integer;
    Begin
       N1:=$N_H$; error3b:=false;suspect $1_{3b}$:=nil;
       suspect $2_{3b}$:=nil;nodecount:=0;m:=count($N_H$);
       repeat
          N2:=$f_2$(N1);nodecount:=nodecount+1;
          If (Id(N2)=Id(N1))
             then N1:=N2
             else error3b:=true
       until(N2=$N_H$) or (error3b) or (nodecount $\geq$ m+1);
       If error3b then
          begin
             suspect $1_{3b}$:N1;
             suspect $2_{3b}$:=N2
          end;
end;
```

Assertion 4 :

The following procedures check for the validity of assertions 4a and 4b. Traversal using one of the two pointer fields is done over the entire structure and a count of the nodes encountered is compared with the stored count for any mismatch.

```
Procedure 4a($N_H$:Node);
   Global Var Error 4a: Boolean;
   Var       N1,N2:Node;m,nodecount:Integer;
    {checks whether the stored count matches with the
                                        number of nodes}
   Begin
        N1:=$N_H$; nodecount:=0; error4a:=false;
        m:=count($N_H$);
        repeat
           N2:=$f_1$(N1); nodecount:=nodecount +1;
           If (N2=$N_H$) then error4a:=nodecount$\neq$m+1
        until(nodecount $\geq$ m+1)or(error4a);
        If(nodecount$\geq$m+1)then error4a:=$N_2 \neq N_H$
   end;
Procedure 4b($N_H$:Node);
   Global Var Error4a:Boolean;
   Var         N1,N2:Node;m,nodecount:Integer;
{Check actual number of nodes, using traversal via $f_2$}
```

```
        Begin
            N1:=$N_H$; nodecount:=0, error4b:=false;
            m:=count ($N_H$);
            repeat
                N2:=f2(N1); nodecount:=nodecount+1;
                If(N2=$N_H$) then error4b:=nodecount$\neq$m+1,
            until (nodecount $\geqslant$ m+1) or (error4b);
            If (nodecount$\geqslant$ m+1) then error4b:=N2$\neq N_H$
end;
```

REFERENCES

1. Black J.P., Taylor D.J., and Morgan D.E., 'A compendium of robust data structures', Dig. 11th Annual Int.Symp. on Fault-Tolerant Comput., Portland, ME, June 24-26, 1981, pp. 129-131.

2. Earley J., 'Towards an understanding of data structures', Comm. ACM, vol. 14, No.10 (Oct. 1971), pp 617-627.

3. Fry J.P., and Sibley E.H., 'Evolution of Data Base Management Systems', Computing Surveys, vol.8, No.1 March, 1976), pp. 7-42.

4. Knuth D.E., 'The Art of Computer Programming': Volume 1 - Fundamental Algorithms', Addison-Wesley, Reading, Mass. 1979, pp. 319-322.

5. Lockemann P.C. and Knutsen W.D., 'Recovery of Disk contents after system failure', Comm. ACM, vol.11, No.8, (Aug. 1968), p. 542.

6. Taylor D.J., Black J.P., and Morgan D.E., 'Redundancy in Data Structures: Improving Software fault tolerance', IEEE Trans. Software Eng., vol.SE-6, No.6, (Nov. 1980), pp. 585-594.

7. Taylor D.J., Black J.P., and Morgan D.E., 'Redundancy in data structures: some theoretical results', IEEE Trans. Software Eng., vol. SE-6, No.6 (Nov. 1980), pp. 595-602.

8. Taylor D.J., and Black, J.P., 'Principles of Data Structure Error Correction', IEEE Trans. Computers, vol. C-31, No.7, (July 1982), pp. 602-608.